Calculating Smarandache Function in Parallel

Sabin Tabirca Tatiana Tabirca and Kieran Reynolds
Department of Computer Science
UCC, Cork, Ireland
Email: {s.tabirca, t.tabirca, kpr1}@cs.ucc.ie

Laurence T. Yang
Department of Computer Science
St. Francis Xavier University
Antigonish, Canada
Email: lyang@stfx.ca

***Abstract*— This article presents an efficient method to calculate in parallel the values of the Smarandache function $S(i)$, $i = 1, 2, ..., n$. The value $S(i)$ can be sequentially found with a complexity of $\frac{i}{\log i}$. The computation has an important constraint, which is to have consecutive values computed by the same processor. This makes the dynamic scheduling methods inapplicable. The proposed solution is based on a *Balanced Workload Block Scheduling* method. Experiments show that the method is efficient and generates a good load balance.**

I. Introduction

The Smarandache function [7] is a relatively new function in Number Theory and yet there are already a number of algorithms for its computation. It is the intention of this article to develop an efficient algorithm to compute in parallel all the values $\{S(i),\ i = 1, 2, ..., n\}$. This is an important problem often occurring in real computation for example in checking conjectures on S.

To begin, the Smarandache function [7] $S : N^* \to N$ is defined as

$$S(n) = \min\{k \in N \mid n | k!\} \tag{I}$$

An important property of this function is given by the following:

$$(\forall a, b \in N^*)(a, b) = 1 \Rightarrow S(a \cdot b) = \max\{S(a), S(b)\}. \tag{2}$$

Expanding on this, it is clear that

$$S(p_1^{k_1} \cdot ... \cdot p_j^{k_j}) = \max\{S(p_1^{k_1}), \ldots, S(p_j^{k_j})\}. \tag{3}$$

Therefore, when trying to evaluate the value of the function at n it is possible to use the prime decomposition of n to reduce the computation. Equation (I) gives a simple formula for $S(p^k)$:

$$k = \sum_{i=1}^{l} d_i \cdot \frac{p^i - 1}{p - 1} \Rightarrow S(p^k) = \sum_{i=1}^{l} d_i \cdot p^i. \tag{4}$$

There have been several studies to show the connection between the function S and prime numbers. It has been proven by Ford [2] that the values of S are almost always prime, satisfying

$$\lim_n \frac{|\{i \le n :\ S(i)\ prime\}|}{n} = 0. \tag{5}$$

Several sequential methods to compute the Smarandache function have emerged since its initial definition in 1980. Ibstedt [3], [4] developed an algorithm based on Equations (3) and (4) without any study of the complexity of this algorithm. Later, Power et.al. [6] analyzed this algorithm and found that the complexity is $O(\frac{n}{\log n})$. The U Basic implementation that was used by Ibstedt has proved to be efficient and useful especially for large values of n. Subsequently, Tabirca [9] studied a simple algorithm based on Equation I by considering the sequence $x_k = k!\ \mod\ n$. This proves a rather inefficient computation that is impractical for large values of n. It was shown that the computation has a complexity of $O(S(n))$. However, studies [I0], [1I] and [5] find that the average complexity of this algorithm is $O(\frac{n}{\log n})$.

```
static long Value (long p, long k) {
      long l, j, value=0;
      long d1[] = new long [1000];
      long d2[] = new long [1000];
      d1[0]=1;d2[0]=p;
      for(int l=0;d1[l]<=k;l++){
           d1[l+1]=1+p*d1[l];
           d2[l+1]=p*d2[l];
      }
      for(l--,j=l;j>=0;j--){
          d=p/d1[j];
          p=p%d1[j];
          value+=d*d2[j];
      }
     return value;
}
```

Fig. 1. The procedure for $S(p^k)$.

A. An Efficient Sequential Algorithm

Performing the computation of the Smarandache function can be done sequentially by developing an algorithm based on Equations (3) and (4). Clearly, if an efficient method to calculate the function on a prime power exists, it is then easy to extend this to the remaining integers. It was this that prompted Ibstedt to develop an algorithm for the computation of the Smarandache function. This algorithm will be briefly examined in this section.

In Equation (4) $(d_l, d_{l-1}, \ldots, d_1)$ is the representation of k in the generalized base $1, \frac{p^2-1}{p-1}, \cdots, \frac{p^l-1}{p-1}$ so that $(d_l, d_{l-1}, \ldots, d_1)$ is the representation of $S(p^k)$ in the generalized base $p, p^2, \ldots, p^l$. This gives a relationship between

Proceedings of the ISPDC/HeteroPar'04
0-7695-2210-6/04 $20.00 © 2004 IEEE

```
public static long S (final long n) {
     long d, valueMax=0, s=-1;
     if (n==1) return 0;
     long p[] = new long [1000];
     long k[] = new long [1000];
     long value[] = new long [1000];
     for(d=2;d<n;d++)
       if (n % d == 0){
          s++;p[s]=d;
          for(k[s]=0;n%d==0;k[s]++,n/=d);
          value[s]=Value(p[s],k[s]);
       }
    for(j=0;j<=s;j++)
        if (valueMax<value[j])
          valueMax=value[j];
    return valueMax;
}
```

Fig. 2. The procedure for $S(p^k)$.

p^k and $S(p^k)$. With this it is possible to write a method to calculate the Smarandache function on a prime power. Once this is in place, it is then possible to calculate the function on any integer.

Note that a prime decomposition algorithm is needed for the computation of the Smarandache function which, for these purposes, can be a simple trial division algorithm. Once a prime decomposition of $n = p_1^{a_1} \cdot ... \cdot p_j^{a_j}$ is available Equation (3) gives $S(n) = \max\{S(p_1^{k_1}), \ldots, S(p_j^{k_j})\}$. This is described in Figure 2.

II. Computing in Parallel

While performing the calculation of $S(n)$ in parallel it is possible, see Power et.al. [6], that is not the purpose of this article. Instead, it is desired that the computation of $\{S(i), i = 1, 2, ..., n\}$ be performed in parallel. Let us suppose that this is done by the **doall** loop

```
do par i=1,n
   calculate S(i)
end do
```

which is computed on a parallel machine with p processors $P_1, P_2, ..., P_p$. The value $S(i)$ is found sequentially by calling the function S from Figure 2 and this is done with a workload of $w_i = \frac{i}{\log i}$, $i = 2, 3, ..., n$. An important requirement of this computation is to have consecutive iterations computed by the same processor. This often occurs when it is needed to check conjectures involving consecutive terms of S.

Computing the above doall loop is a classical scheduling problem in parallel computation. Scheduling methods find a mapping of the iterations onto the processors. This means that the set of indices $\{1, 2, ..., n\}$ is partitioned into p sets $\{S_j,\ j = 1, 2, ..., p\}$. Scheduling methods are classified into two main categories depending on when the partition is found. *Static Scheduling Methods* generate the partition during compile time while *Dynamic Scheduling Methods* find it during run time. The main advantage of the latter is that they can detect when a processor becomes idle and assign iterations to it. Studies have shown that *Dynamic Scheduling Methods* achieve a good load balance of the workloads. However, they produce small scheduling overheads.

On the other hand *Static Scheduling Methods* do not give any scheduling overheads but they usually give a poor imbalance of the workloads. The simplest way to schedule statically the iterations is to assign $\frac{n}{p}$ consecutive iterations to each processor. In this case processor j receives the iterations $\frac{(j-1)\cdot n}{p}+1, \frac{(j-1)\cdot n}{p}+2, \cdots, \frac{j\times n}{p}$. This method, which is called *Uniform Block Scheduling* gives good load balance when the workloads $\{w_1, w_2, ..., w_n\}$ are similar. When the workloads increase or decrease the method is clearly inefficient because there is one processor that gets all the biggest n/p workloads. *Cyclic Scheduling* corrects this inconvenience by distributing the iterations in a cyclic fashion so that two consecutive big workloads are not assigned to the same processor. The method allocates to processor j the iterations $\{j,\ j+p,\ j+2\cdot p, \cdots, j+\left[\frac{n-j}{p}\right]\cdot p\}$. Certainly, cyclic scheduling offers an efficient load balancing when the workloads decrease or increase.

Tabirca [13] proposed a recent static scheduling method named *Balanced Workload Block Scheduling (BWBS 1)*. This is a block scheduling in which processor j receives the consecutive iterations $\{l_j, l_j + 1, ..., h_j\}$ so that its workload is balanced. Hence, the scheduling is defined by the lower and upper bounds $\{(l_j, h_j),\ j - 1, 2, ..., p\}$ so that

$$l_1 = 1,\ h_p = n,\ l_j = h_{j-1} + 1,\ j = 2, .., p.$$

Suppose that there is an estimation or a formula for the workloads $\{w_1, w_2, ..., w_n\}$. Therefore, the workload for the entire loop is given by $w = \sum_{i=0}^{n} w_i$ and the average workload per processor is given by

$$\bar{W} = \frac{1}{p} \cdot \sum_{i=0}^{n} w_i.$$

Clearly, good scheduling should give bounds (l_j, h_j) for processor j such that

$$\sum_{i=l_j}^{h_j} w_i \simeq \frac{1}{p} \cdot \sum_{i=1}^{n} w_i := \overline{W}, \quad \forall j = 1, 2, ..., p.$$

To evaluate bounds for the computation, two functions are needed. Firstly, by extending the inferior part function, define

$$f_{[]}(x) = k \Leftrightarrow f(k) \leq x < f(k+1). \tag{6}$$

Tabirca et al [13], show that if both $f_{[]}$ and the function $f(h) = \sum_{k=1}^{h} w_k$ exist or can be calculated then the upper bounds are given by

$$h_j = f_{[]}(\overline{W} + f(h_{j-1})), j = 1, 2, ..., p. \tag{7}$$

However, the method can still be applied when there are not formulas for these two functions. In this case a pre-processing

step is required to calculate the average workload $\overline{W}$ and the upper bounds $\{h_j,\ j = 1, 2, ..., j\}$ using

$$h_j = h \Leftrightarrow \sum_{i=l_j}^{h} w_i \leq \overline{W} < \sum_{i=l_j}^{h+1} w_i. \quad (8)$$

The pre-processing step however gives a scheduling overhead of $O(\frac{n}{p})$.

Tabirca [12] proposed an improvement on this method considering the partial sum that is closest to $j \cdot \overline{W}$

$$h_j = h \Leftrightarrow \sum_{i=1}^{h} w_i \leq j \cdot \overline{W} < \sum_{i=1}^{h+1} w_i. \quad (9)$$

In this case the upper bounds are given by

$$h_j = f_{[]}(j \cdot \overline{W}), j = 1, 2, ..., p. \quad (10)$$

This *Balanced Workload Block Scheduling (BWBS 2)* method has been proven to be marginally better than the initial one.

For the loop we study the workloads $w_1 = 0$, $w_i = \frac{i}{\log i}$, $i = 2, 3, ..., n$ increase so that the *Unifrom* scheduling does not give an efficient solution. Certainly, the *Dynamic Scheduling* or *Cyclic* methods can be applied to obtain a better load balance. Unfortunately, they are not suitable because the processors do not get consecutive iterations. Therefore, the *Balanced Workload Block Scheduling* methods remain to give an efficient solution for our problem. Since the sum $\sum_{i=0}^{n} \frac{i}{\log i}$ does not have a formula or a simple approximation, the pre-processing step must be applied to achieve the scheduling bounds.

	P_1	P_2	P_3	P_4
Uniform	225.25	611.45	971.45	1318.54
BWBS 1	787.81	780.78	777.61	785.18
BWBS 2	782.65	781.52	782.05	782.34

TABLE 1
EXECUTION TIMES ON PROCESSORS.

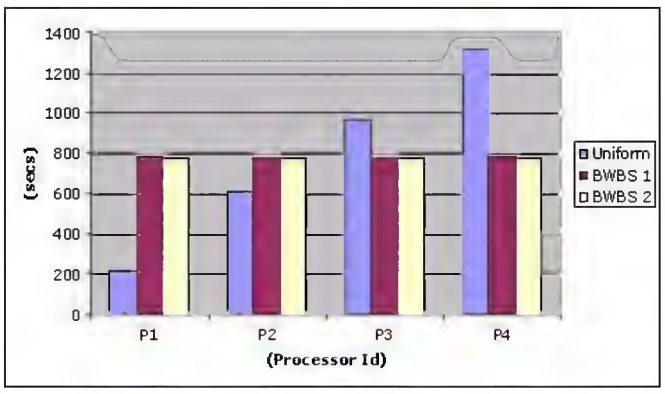

Fig. 3. Execution Times on Processors.

III. EXPERIMENTAL RESULTS

In this section some experimental results are outlined to show how the problem is solved in parallel. The computation has been performed on a 100 node Beowulf cluster. The machine consists of 50 Dell Poweredge 1655MC servers each of them with a dual Pentium III processor (1.26GHz, 512K cache, 1 GB of RAM).

The *Uniform* and *Balanced Workload Block Scheduling* methods are considered to schedule the loop iterations. To test these approaches, we generate $S(i) \forall i \leq 100,000,000$ and check whether the equation $S(i) = S(i+1)$ has solutions. This is an old standing conjecture that has been checked by Ibstedt [3] for all the numbers $i \leq 1,000,000$. It has been conjectured that the equation has no solutions.

	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Uniform	2925.62	2088.53	1215.34	718.54	434.92	271.82
BWBS 1	2925.72	1543.87	787.81	412.18	237.61	148.50
BWBS 2	2925.68	1524.25	782.65	397.34	210.53	131.58

TABLE II
VARIATION OF EXECUTION TIMES.

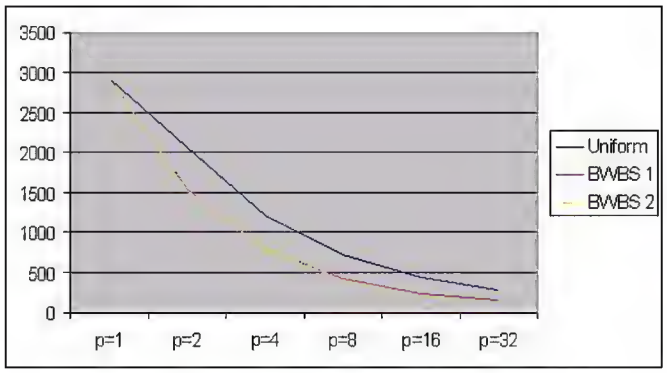

Fig. 4. Variation of Execution Times.

The first test presents the workload distribution on processors. For that the loop is scheduled on 4 processors and the computation time on each processor is measured. This gives an estimation of workload balance of each method. Table 1 and Figure 3 show the variation of these execution times. It can be seen that the *Uniform Scheduling* method generate a huge imbalance where processor 4 is more than 6 times loaded than processor 1. On the other hand the *BWBS* methods give an efficient load balance with a marginal advantage for the second one. The desired effect of *BWBS* balancing, which is to get times on each processor as 'nearly' equal as possible is clearly visible from the diagram.

The second test investigates the variation of the overall execution times when the number of processors vary. The above loop has been run using $p = 1, 2, 4, 8, 16$ processors. The variation of the execution times is presented in Table 2. Examining Figure 4 shows, unsurprisingly, that the *BWBS* bounds offer not only the best balance but also the quickest computation.

IV. Final Conclusion

This article has presented how the values of the Smarandache function can be found in parallel. It has been required that consecutive values have to be calculated by the same processor. This has restricted the scheduling methods that could have been used for the computation. A variation of *Balanced Workload Block Scheduling* has been used to achieve efficient computation. That has been possible only because the number of operations to compute $S(i)$ is known.

Based on this method several conjectures from Smarandache's Open Problem list [8] have been verified for all values up to 1,000,000,000 using the BWBS scheduling. Unfortunately, no counterexample has been found to disprove any of them so that we can say that they are true at least for all the values under one billion. This type of computation can also be used to generate in parallel the values of some other Number Theory functions e.g. Erdos' or Euler's.

References

[1] E. Bach and J. Shallit, *Algorithmic Number Theory*, MIT Press, Cambridge, Massachusetts, USA, 1996.

[2] G. Ford, An Asymptotic Evaluation of the Smarandache Function, *Smarandache Notion Journal*, 11, No.1-2-3, 2000, 45-61.

[3] H. Ibstedt, *Surfing on the Ocean of Numbers - a few Smarandache Notions and Similar Topics*, Erhus University Press, New Mexico, USA, 1997.

[4] H. Ibstedt, *Computational Aspects of Number Sequences*, American Research Press, Lupton, USA, 1999.

[5] F. Luca, The average Smarandache Function, *Smarandache Notion Journal*, 12, No.1-2, 2001, 134-142.

[6] D. Power, S. Tabirca and T. Tabirca, Java Concurent Program for the Smarandache Function, *Smarandache Notion Journal*, 12, No.1-2, 2001, 121-132.

[7] F. Smarandache, A Function in number theory, *Analele Univ. Timisoara*, XVIII, 1980, 142-156.

[8] F. Smarandache, *Only Problems ... Not Solutions*, Xiquan Publishing House, Phoenix-Chicago, 1993.

[9] S. Tabirca and T. Tabirca, Some Computational Remarks on the Smarandache Function, *Proceedings of the First International Conference on the Smarandache Type Notions*, 1997, Craiova, Romania.

[10] S. Tabirca and T. Tabirca, Some upper bounds for Smarandache's function, *Smarandache Notions Journal*, 8, 1997, 205-211.

[11] S. Tabirca and T. Tabirca, Two new functions in number theory and some upper bounds for Smarandache's function, Smarandache Notions Journal, 9, No. 1-2, 1998, 82-91.

[12] T. Tabirca and S. Tabirca, A New Equation for the Balanced Loop Scheduling Based on the Smarandache Inferior Part Function to Loop Scheduling, *Proceedings of the 2nd International Conference on Smarandache Types Notion*, 2001, Romania.

[13] T. Tabirca, L. Freeman, S. Tabirca and T.L.Yang, A Static Workload Balance Scheduling Algorithm, *Proceedings of the 2nd Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications (PDSECA 2001)*, April 2001, San Francisco, USA.

0-7695-2210-6/04 $20.00 © 2004 **IEEE**